VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Analýza a srovnání knihoven pro persistenci dat v .NET

Analysis and comparison of libraries for data persistence in .NET

2014 Ondřej Sobek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Ondřej Sobek**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Analýza a srovnání knihoven pro persistenci dat v .NET
Analysis and Comparison of Libraries for Data Persistence in .NET

Zásady pro vypracování:

1. Najděte aktuální knihovny (a jejich rozšíření) určené pro persistenci dat v .NET a popište jejich teoretické i praktické vlastnosti. Tyto vlastnosti ukažte na příkladech.
2. Proveďte srovnání a kategorizaci těchto knihoven podle některých vlastností (například typu, účelu, aj.).
3. Popište nejčastější scénáře použití persistence dat a doporučte vhodné knihovny. Svoje doporučení zdůvodněte.

Seznam doporučené odborné literatury:

Dle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Jakub Macek**

Datum zadání: 18.11.2011

Datum odevzdání: 07.05.2013

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2013 ..... Tober

Chtěl bych poděkovat vedoucímu práce Ing. Jakubu Mackovi za cenné rady a doporučení.

Abstrakt

Tento dokument prakticky i teoreticky popisuje práci s některými známými .NET knihovnami pro persistenci dat. Práce je rozdělena do čtyř hlavních částí - nejdříve knihovny, které k ukládání dat používají relační model, dále knihovny využívající dokumentové (NoSQL) databáze a nakonec také jeden zástupce objektového přístupu. Na konci každé části jsou knihovny porovnány a doporučeny dle možných preferencí čtenáře. V závěru práce jsou popsány doporučené scénáře persistence pro každý typ databáze. Text je doplněn ukázkami kódů, jež popisují některé ze zajímavějších schopností knihoven.

Klíčová slova: .NET, Dapper, Entity Framework, NHibernate, RavenDB, Redis, Eloquera, bakalářská práce, persistence dat, databáze

Abstract

This document both practically and theoretically describes work with some of the famous .NET libraries for data persistence. The task is divided into three main parts - firstly libraries, which are using relational databases to store their data, then libraries which make use of the document (NoSQL) databases and lastly one delegate of the object oriented approach. At the end of each part libraries are compared and recommended based on possible user preferences. Recommended persistence scenarios are described at the end of the document. The text is complemented with code examples on how to use some of the more interesting abilities of each library.

Keywords: .NET, Dapper, Entity Framework, NHibernate, RavenDB, Redis, Eloquera, bachelor's thesis, data persistence, database

Seznam použitých zkratek a symbolů

XML	–	Extensible Markup Language
SQL	–	Structured Query Language
RAM	–	Random-Acces Memory
ORM	–	Objektově-relační mapování
CRUD	–	Create, Read, Update, Delete
POCO	–	Plain Old CLR
CLR	–	Common Language Runtime
OOP	–	Objektově orientované programování
LINQ	–	Language Integrated Query
API	–	Application programming interface
EF	–	Entity Framework
PMC	–	Package Manager Console
NH	–	NHibernate
JSON	–	JavaScript Object Notation
HQL	–	Hibernate Query Language
IIS	–	Internet Information Services
ACID	–	Atomicity, Consistency, Isolation, Durability
DTC	–	Distributed Transaction Coordinator
OSS	–	Open source software
JVM	–	Java Virtual Machine
OLTP	–	Online Transaction Processing
RDBMS	–	Relational Database Management System
CAD	–	Computer Aided Design

Obsah

1 Úvod **3**

2 ORM knihovny **4**
2.1 Dapper 4
2.2 Entity Framework 8
2.3 NHibernate 14
2.4 Srovnání ORM knihoven 19

3 Dokumentové knihovny **21**
3.1 RavenDB 21
3.2 Redis 25
3.3 Srovnání NoSQL knihoven 28

4 Objektová databáze **30**
4.1 Eloquera 30

5 Scénáře persistence **33**
5.1 Relační databáze 33
5.2 Dokumentové databáze 33
5.3 Objektové databáze 34

6 Závěr **35**

7 Reference **36**

Přílohy **36**

Seznam výpisů zdrojového kódu

1 Dapper - Transakce 5
2 Dapper - Multi mapping 6
3 Dapper - Uložená procedura 6
4 Dapper - Multiple results 7
5 Entity Framework - Lazy loading 11
6 Entity Framework - Eager loading 11
7 Entity Framework - Transakce 12
8 Konfigurační soubor NHibernate 15
9 NHibernate - Lazy loading 16
10 NHibernate - dotazy 17
11 NHibernate - Transakce 17
12 RavenDB - Transakce 22
13 RavenDB - Lazy loading 23
14 RavenDB - Vyhledávání 24
15 Redis transakce 26
16 Eloquera - dotazování 31
17 Eloquera - connection string 31

1 Úvod

Již poměrně dlouhou dobu je na počítačové programy kladen požadavek, aby si data uchovávaly i po svém ukončení a znovunačtení. Jelikož spuštěné programy běží v RAM paměti, jejíž obsah se po ukončení programu nebo vypnutí počítače maže, musí ovládat nějakou formu persistence (zachování) svých dat. Taková data jsou nejčastěji ukládána do databází. Aby byl proces ukládání dat zjednodušen, vznikly jisté knihovny či frameworky, které programátorům tuto práci usnadňují. Pochopitelně existuje více způsobů, jakými lze data ukládat - relační, dokumentové, grafové databáze, atd. V rámci této práce jsem se zaměřil na nejčastěji používané relační (kapitoly Dapper, Entity Framework, NHibernate), NoSQL (kapitoly RavenDB, Redis) a jednoho zástupce objektových databází (kapitola Eloquera). Dle zadání jsem se všemi knihovnami pracoval na .NET platformě.

2 ORM knihovny

V této časti se budeme zabývat ORM knihovnami, tedy těmi, které ukládají data do relačních databází. Dnešní programy pracují s daty ve formě objektů, jenže takovéto objekty nelze jen tak uložit do relační databáze. Je potřeba zajistit platformu, která programová a tedy čistě objektová data "přeloží"do takového formátu, jež je možno uložit do databáze. Tento proces je do jistý míry stálý, proto vznikly frameworky, které tuto operaci usnadňují.

Pokud chcete spustit přiloženou solutionu do Visual Studia, musíte kromě správné instalace jednotlivých knihoven také zajistit, aby na daném počítači běžela instance Microsoft SQL Serveru, já jsem využil SQL Server 2008R2 (verze 10.50). Na server je pochopitelně nutné nahrát databázi, s níž budeme pracovat - skripty jsem umístil do složky `DB_scripts`. Jméno mojí instance SQL Serveru je BP a databáze nesou názvy Fabrics a Bank. Není nutné toto dodržet, nicméně to doporučuji, jelikož by jinak bylo zapotřebí upravit connection stringy v jistých třídách.

2.1 Dapper

Dapper je volný open source software vyvinutý komunitou kolem webového fóra StackOverflow. Myšlenka této knihovny spočívá v rychlém generování objektů z výsledků SQL dotazů, čehož dosahuje díky cacheování informací u každého dotazu a minimální režii. Dapper neřeší vztahy mezi objekty a nedokáže automaticky generovat žádné SQL, proto jej tvůrci označují jako micro-ORM. [1]

2.1.1 Instalace

Dapper je pouze jeden soubor, který stačí referencovat v daném projektu a lze s ním začít ihned pracovat. Je možné jej získat na adrese http://code.google.com/p/dapper-dot-net/, nicméně pro pohodlnost doporučuji k instalaci použít balíčkovací systém Visual Studia - NuGet. Pokud nepoužíváte Visual Studio 2012, můžete NuGet stáhnout a doinstalovat - http://nuget.org/. Stačí v něm vyhledat klíčové slovo Dapper a nainstalovat první vyhledanou položku - Dapper dot net (občas se vyhledávání buguje a je potřeba frázi vyhledat znovu). NuGet nově nainstalovanou součást automaticky i zareferencuje do projektu, tedy není nutné provádět další úkony a je možné s Dapperem začít ihned pracovat. Existují i jistá rozšíření Dapperu jako např. Dapper-Extensions, která přinášejí zjednodušení CRUD operací a několik dalších vylepšení.

2.1.2 Technologie

Jelikož je Dapper ve své podstatě spíše jen objektový mapper, nepodporuje různé technologie velkých frameworků jako lazy loading nebo sledování změn nahraných objektů v rámci nějaké kontextové třídy. Přesto se určitá funkcionalita nad rámec běžných CRUD operací zajistit dá.

Transakce

Jsou to sekvence databázových operací, které se musí buď provést úplně všechny, anebo žádná, čímž bude databáze vždy zanechána v korektním stavu. [5] Typickým příkladem využití transakcí je bankovnictví. Při převodu peněz mezi dvěma účty se úspěšně musí provést vícero operací - odečtení částky z účtu A, přičtění na účet B, případně ještě nějaké další úkony. Kdyby se např. peníze odečetly z prvního účtu, ale už se nepřičetly na druhý, došlo by k nekorektnímu stavu databáze (peníze nemohou jen tak zmizet). Tento problém lze vyřešit využitím transakcí viz. příklad níže.

Dapper sám o sobě transakce nijak implicitně nepodporuje, nicméně je možné využít ADO.NET třídu TransactionScope. Navíc je pro správnou funkčnost programu nezbytné mít v systému zaplou službu Koordinátora distribuovaných transakcí (Distributed Transaction Coordinator).

```
using (var transactionScope = new TransactionScope())
{
    using (var sqlConnection = new System.Data.SqlClient.SqlConnection(Connectionstring))
    {
        try
        {
            sqlConnection.Open();
            var accountFrom = helper.getAccount(idfrom);
            helper.updateFromAccount(accountFrom, amount);
            var accountTo = helper.getAccount(idto);
            helper.updateToAccount(accountTo, amount);
            helper.insertTransaction(idfrom, idto, amount);
            transactionScope.Complete();
            sqlConnection.Close();
            Console.Write("Vsechny operace uspesne dokonceny.");
        }
        catch (Exception)
        {
            Console.Write("Chyba, rollback. ");
        }
    }
}
```

Výpis 1: Dapper - Transakce

Ukázka z výpisu 1 převede peníze z jednoho účtu na druhý a zároveň uloží záznam o provedení transakce, avšak pouze v případě, že v pořádku proběhnou všechny operace. Kdyby něco bylo v nepořádku (např. neexistující ID účtu), všechny úkony se zruší a databáze bude navrácena do posledního korektního stavu.

Na první pohled kód vypadá poměrně přímočaře, avšak jen proto, že jeho většina je ukrytá ve funkcích jako `getAccount()`, `insertTransaction()` apod. Narozdíl od velkých frameworků Dapper sám o sobě neumí např. materializovat objekt včetně kolekce jeho potomků, proto je nutné velkou část kódu včetně SQL dotazů psát ručně. To poskytuje výhodu dobré kontroly nad SQL směřovaným na databázi, na druhou stranu někteří vývojáři by raději byli od SQL oprostěni a věnovali se místo něj samotnému programu.

Multi mapping
Dapper vývojáři umožňuje namapovat jeden řádek na vícero objektů.

```
sqlConnection.Open();
var query = "select␣a.AccountId,␣a.Balance,␣c.CustomerId,␣c.Name␣from␣Accounts␣as␣a␣join␣
    Customers␣as␣c␣on␣a.AccountCustomerId␣=␣c.CustomerId";
var accounts = sqlConnection.Query<Account, Customer, Account>(query, (a, c) => { a.Customer =
    c; return a; }, splitOn: "CustomerId");
sqlConnection.Close();
return accounts;
```

Výpis 2: Dapper - Multi mapping

Výpis 2 je výňatkem z metody `getAllAccounts()`, jež vrací všechny účty a jméno jejich majitele. V prvé řadě je nutné napsat samotný SQL dotaz - tuto abstrakci Dapper zkrátka neprovádí. Zajímavější je metoda `Query()`, která slouží k získávání dat z databáze a následnému plnění objektů. V tomto případě dotaz nevrací pouze účet, ale i jeho majitele, proto je nutné do prvního parametru metody vložit oba. Ovšem aby toto fungovalo, Dapper musí vědět, jak má spojit účet s jeho majitelem (z toho důvodu má doménová třída Account property Customer, která neexistuje jako sloupec v databázi, ale bude naplněna daty majitele, jež byl s účtem spojen v dotazu). Ve výčtu entit se znovu objevuje účet (Account), čímž Dapper zjistí, který objekt má přesně vracet. Po vloženém dotazu pak následuje funkce, která Dapperu říká, jak namapovat účet a jeho majitele do jednoho účtu pro každý řádek vrácený daným dotazem. Nakonec na řadu přichází parametr `splitOn()`, jež specifikuje klíč, kterým jsou tabulky spojeny (v podstatě bod rozdělení). Implicitně je tato funkce prováděna pro sloupec s názvem Id nebo ID, takže pokud jsou klíče v databázi pojmenovány přesně takto, není nutné `splitOn()` používat. Občas vznikne potřeba rozdělovat na více sloupcích - v takovém případě stačí všechny sloupce zapsat jako parametry metody `splitOn()` a oddělovat čárkou **bez mezery**.

Uložené procedury
Jsou shluky zpravidla T-SQL nebo PL/SQL příkazů zkompilovaných v rámci jednoho vykonávacího plánu. Všechny moderní DMBS je podporují, jelikož nasazení uložených procedur může přinášet značné výhody - např. snížení síťové zátěže, bezpečnost, snadnější údržbu nebo zvýšený výkon. [2]

```
var customer = sqlConnection.Query<Customer>("getUserById", new { @p_id = customerid },
    commandType: CommandType.StoredProcedure).FirstOrDefault();
Console.WriteLine("{0}␣{1}␣{2}", customer.Name, customer.City, customer.Email);
```

Výpis 3: Dapper - Uložená procedura

Volání uložené procedury je v Dapperu podobné klasickému dotazování prostřednictvím metody `Query()`. Místo dotazu Dapperu postačí název uložené procedury v databázi, zadaný parametr (pokud procedura nějaký očekává) a nastavení `commandType`, z něhož Dapper pozná, co přesně má dělat. Kód v ukázce spustí proceduru s předaným Id uživatele, načež procedura vrátí podrobnosti o daném účtu jakožto property objektu, se kterým je možné v programu pracovat. V případě potřeby Dapper také umožňuje použití dynamických parametrů.

Multiple results
Dapper také dokáže zpracovat vícero množin výsledků v rámci jednoho dotazu:

```
var sql =@"select␣*␣from␣Customers␣where␣CustomerId␣=␣@id
select␣*␣from␣Accounts␣where␣AccountCustomerId␣=␣@id
select␣*␣from␣Transactions␣where␣TransactionFromId␣=␣@id␣or␣TransactionToId␣=␣@id";
    using (var multi = sqlConnection.QueryMultiple(sql, new { id = customerid }))
    {
        var customer = multi.Read<Customer>().Single();
        var accounts = multi.Read<Account>().ToList();
        var transactions = multi.Read<Transaction>().ToList();
    }
```

Výpis 4: Dapper - Multiple results

Toto je výhodné zejména pro snížení počtu spojení na databázi a zlepšení výkonu aplikace.

2.1.3 Dapper - shrnutí

Vznik Dapperu se datuje k 14.4. 2011, kdy byla zveřejněna první verze 1.0.0. Důvodem jeho vývoje bylo to, že provoz kolem webového fóra StackOverflow narostl do takových rozměrů, že původní ORM už nadále nebylo schopné data zpracovávat v rozumném čase. Většinu zátěže stránek tvořilo obrovské množství jednodušších dotazů, což byla hlavní příčina vzniku Dapperu - mapování parametrizovaného SQL na business objekty.

Jak už jsem psal na začátku kapitoly, tato knihovna je označována jako micro-ORM. Je to z toho důvodu, že se v podstatě jedná o specializovaný mapper, který vůbec neřeší objektové vztahy, sledování změn nahraných dat, lazy loading (avšak lze provést vlastní implementaci skrze třídu Lazy<T>), automatické generování SQL, apod. Veškerá automatizace ustoupila jednoduchosti a rychlosti. Existují však jistá rozšíření Dapperu, která určitým způsobem zlepšují či rozšiřují jeho funkcionalitu. Z těchto bych vyzdvihl dvě:

- Dapper-Extensions - poskytuje helpery pro CRUD operace
- Dapper dot net async - přidává možnost asynchroního dotazování

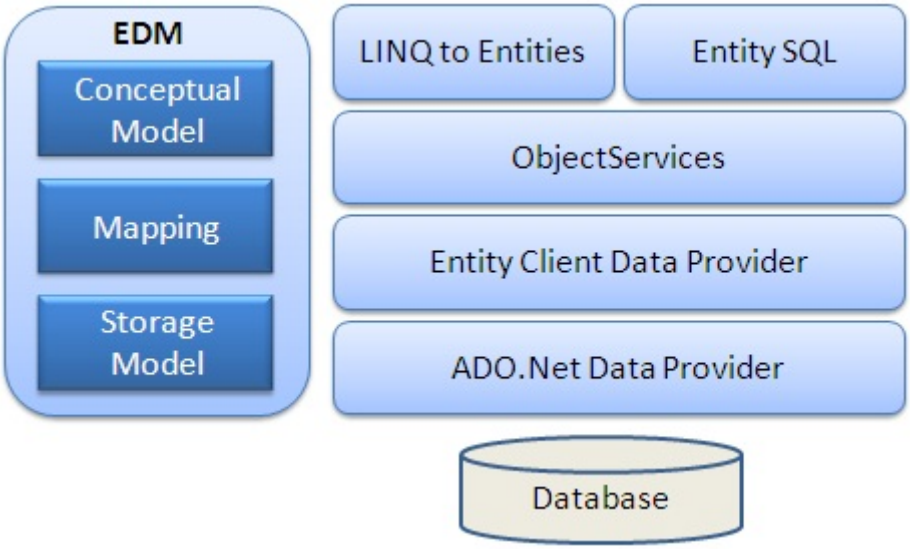
V současné době se Dapper nachází ve verzi 1.26 a je nasazen na projektech Stack Overflow, xpfest.com, helpdesk-software, worldcitycard, roadmap. Lze tedy prohlásit, že se jedná o stabilní a osvědčenou knihovnu.

Ideálním případem užití by bylo nasazení do situace, která vyžaduje materializaci spousty doménových objektů, přičemž se příliš neočekává využití technologií velkých frameworků (potenciálně např. webové fórum).

2.2 Entity Framework

Entity Framework je Microsoftem podporované řešení ORM pro .NET. Současný stabilní build se nachází ve verzi 6.1. Podobně jako další frameworky i tento ulehčuje vývojáři práci tím, že zajišťuje konverzi mezi programovými (objektovými) a relačními daty v databázi. Narozdíl od Dapperu je Entity Framework poměrně robustní ORM a nabízí některé zajímavé funkce jako např. Model First, Database First, Code First development, nativní LINQ podporu atd. Nedělá mu tedy problém vytvořit model a na základě něj pak databázi, nebo naopak vytvořit model podle již existující databáze. [3]

Architektura Funkčnost Entity Frameworku si můžeme ilustrovat obrázkem:



EDM (Entity Data Model): Skládá se ze tří částí - konceptuální model, mapování a model ukládání.

Conceptual Model: Konceptuální model slouží jako grafická reprezentace reality. Obsahuje typy entit, asociace, komplexní typy a další v aplikační doméně.

Storage Model: Definován skrze SSDL (store schema definition language), popisuje databázové schéma.

Mapping: Obsahuje informace o tom, jak je konceptuální model namapován na databázi.

LINQ to Entities: Dotazovací jazyk k psaní dotazů na objektový model. Vrací entity, jež jsou definovány v konceptuálním modelu.

Entity SQL: Další dotazovací jazyk podobný LINQ to Entitties. Je však složitější a vývojáři se jej musí naučit zvlášť.

Object services: Hlavní přístupová brána k databázi. Provádí tzv. materializaci, což je proces konvertování dat vrácených Entity Client Data Providerem na strukturu objektu.

Entity Client Data Provider: Převádí L2E nebo Entity SQL na dotazy jazyka SQL, kterému rozumí daná databáze. Komunikuje s ADO.NET Data Providerem, který čte nebo zapisuje data do databáze.

ADO.Net Data Provider: Tato vrstva komunikuje přímo s databází. Lze využít i providery pro jiné databáze - např. MySQL nebo Oracle.

2.2.1 Instalace

Nejpohodlnějším způsobem je instalace skrze NuGet, stačí vyhledat Entity Framework a nainstalovat jej. Další zajímavou "vychytávkou"jsou tzv. Entity Framework Power Tools. Jedná se o rozšíření Visual Studia, které funguje jako interaktivní návrhář pro Entity Framework. Mezi jeho schopnosti patří např. Reverse Engineer Code First - automaticky vygeneruje POCO třídy a mapování pro existující databázi, což dokáže ušetřit mnoho času, případně zobrazit Entity Data Model.

2.2.2 Technologie

Entity Framework umožňuje hned několik různých metod vývoje - workflow. Vhodnou metodu pro daný projekt si je možno vybrat po zodpovězení základních otázek:

- Je už připravená databáze, nebo se bude vytvářet na základě doménového modelu?
- Chcete model vytvářet v grafickém návrháři, nebo raději psát kód?

1. Code First (nová databáze): Tento způsob vývoje bude vyhovovat zejména lidem, kteří rádi dodržují principy DDD (Domain Driven Design). Umožňuje programátorům soustředit se na doménový model, namísto návrhu databáze a vytváření tříd podle ní. V praxi se tedy nejprve napíší doménové třídy a až při spuštění aplikace vytvoří Code First API novou databázi. Napojení na konkrétní databázový server lze předat parametrem v konstruktoru třídy, respektive vepsáním do konfiguračního souboru aplikace.

2. Model First: Grafická obdoba předchozího workflow. Nejprve se vizuálně navrhne doménový model, na jehož základě jsou následně vygenerovány `C#` třídy. Z těchto tříd pak Entity Framework vytvoří databázi přímo na databázovém serveru.

3. Database First: Jak už název napovídá, tento workflow je vhodné použít v případě, že pro daný projekt již existuje databáze naplněná daty. Stačí zadat spojení na databázi, načež Entity Framework pomocí techniky reverzního inženýrství vygeneruje grafickou reprezentaci doménového modelu (edmx soubor) a z ní pak následně i jednotlivé třídy. Návrhář si poradí i se situací, kdy je v databázi provedena nějaká změna (např. nová tabulka) - je schopný kdykoliv aktualizovat model z databáze.

4. Code First (existující databáze): V tomto workflow se definují doménové třídy a mapování skrze klasické psaní kódu pro již existující databázi. I zde je možno využít nástroje pro automatické generování POCO tříd z databáze.

V souvislosti s Code First přístupy je také potřeba zmínit podporu tzv. **migrací**. Migrace představují pro Entity Framework další nástroj, s jehož pomocí se lze vypořádat se změnami v databázi, případně jej použít k vytvoření nové. Zapnutí migrací vytvoří

uspořádanou množinu kroků, které popisují různé verze databázového schématu. Každý z těchto kroků je známý jako migrace a obsahuje kód, jež popisuje změny v databázi. K lepšímu pochopení výhod migrací je vhodné si popsat tzv. inicializátory databáze pro Entity Framework. Inicializátor databáze je třída, která se stará o vytvoření databáze, tedy tabulek a vazeb podle daného doménového modelu.

Entity Framework standardně disponuje třemi inicializátory:

CreateDatabaseIfNotExists: Tento inicializátor se používá jako výchozí. Vytvoří databázi pouze v případě, že taková neexistuje. Zároveň se vyhýbá jejímu neúmyslnému smazání.

DropCreateDatabaseWhenModelChanges: Tato třída vytvoří novou databázi pouze pokud neexistuje, nebo došlo k neshodě mezi modelem a aktuální formou databáze. Z toho důvodu se obvykle používá v ranných fázích vývoje nebo testování, kdy se model často mění a není kladen důraz na existující databázové záznamy.

DropCreateDatabaseAlways: Při použití tohoto inicializátoru se databáze vždy smaže (ať už existuje, nebo ne) a znovu vytvoří při každém spuštění aplikace, což je vhodné pro testování, kdy je potřeba spouštět aplikaci vždy s novým datasetem.

Další možností je napsat si vlastní inicializátor databáze, čehož lze docílit využitím IDatabaseInitializer rozhraní. V takovém případě je potřeba implementovat metodu InitializeDatabase() a napsat vlastní logiku tvorby databáze.

Tyto inicializátory však nemusí každému vyhovovat - např. chceme přidat novou modelovou třídu a podle ní pouze aktualizovat databázi bez jejího smazání nebo jiné změny. Takovýto problém lze vyřešit pomocí Code First migrací. Ty je možno zapnout příkazem `Enable-Migrations`, který se vloží do Package Manager Console. Spuštění tohoto příkazu v projektu vytvoří novou složku s názvem Migrations, jejíž počáteční obsah bude roven dvěma souborům - třídě Configuration a první uložené migraci s názvem `<timestamp>_InitialCreate` (timestamp je "časová známka"- větší počet vygenerovaných čísel).

Konfigurační třída specifikuje nastavení, která budou migrace používat pro daný kontext. Za zmínku určitě stojí Seed metoda, jež má za úkol tzv. "zasít"databázi novými daty, případně aktualizovat data stávající a je tedy vhodná pro naplnění databáze testovacími daty při odlaďování aplikace.

První migrace s názvem `_InitialCreate` obsahuje změny, které byly na databázi provedeny od jejího vytvoření. Každá další migrace vytváří snapshot databáze, díky čemuž lze jednoduše přecházet mezi různými verzemi. Novou migraci lze vytvořit pomocí příkazu `Add-Migration <jméno>` a následně ji aplikovat na databázi skrze Update-Database (obojí se vkládá do PMC), přičemž po potvrzení druhého příkazu se provedená migrace zaznamená i na úrovni databáze (tabulka `_MigrationHistory`). Pro výpis všech již aplikovaných migrací poslouží příkaz `Get-Migrations`.

Lazy loading

Jedná se o návrhový vzor, u něhož příbuzné objekty (potomci) nejsou automaticky nahrávány zároveň s mateřským objektem, ale až na vyžádání. Jeho vhodné použití může v určitých případech zvýšit efektivitu aplikace.

```
using (var db = new FabricsContext())
{
    db.Configuration.LazyLoadingEnabled = true;
    db.Database.Log = s => System.Diagnostics.Debug.WriteLine(s);
    Console.Write("ID klienta: ");
    Int32 idclient = Convert.ToInt32(Console.ReadLine());
    var client = db.Clients.Where(c => c.ClientId == idclient).FirstOrDefault();
    IList<Order> orders = client.Orders.ToList();
}
```

Výpis 5: Entity Framework - Lazy loading

Tento kód načte klienta zadaného z klávesnice a vypíše jeho jméno a objednávky, které jsou pod příslušným ID uloženy v jiné tabulce. Za zmínku stojí kontextová třída (v tomto případě FabricsContext), což je hlavní třída zodpovědná za interakci s objektovými daty - naplňování objektů daty z databáze nebo ukládání do ní. [4] Je kombinací vzorů Unit of Work a Repository takovým způsobem, aby mohla být použita k dotazování na databázi a shlukovat změny, které budou zapsány zpět na úložiště formou jednotky.

Paremetr `LazyLoadingEnabled` není nutné nastavovat vždy - pokud se pro tvorbu modelu použijí nástroje Entity Frameworku, vygenerovaný kód už tuto vlastnost bude mít zaplou. Nutno zdůraznit, že pro funčnost lazy loadingu je zapotřebí deklarovat jednotlivé property v doménových třídách jako virtuální (v opačném případě bude pro danou property vypnut).

Kód nahoře bude mít za následek odeslání dvou SQL dotazů. Entity Framework nejdříve z databáze získá klienta, a až na požádání aplikace následně odešle druhý dotaz, kde se ptá na objednávky pro danou osobu.

Eager loading

Proti lazy loadingu funguje obráceně - potomci jsou načítáni zároveň s jejich rodiči. V Entity Frameworku jeho použití vynutí příkaz `Include`:

```
var query = db.Clients.Include("Orders").Where(c => c.ClientId == idclient).ToList();
```

Výpis 6: Entity Framework - Eager loading

Tato zjednodušená ukázka dělá ve výsledku to samé jako ta předchozí, ale rozdíl je v interním fungování aplikace. V tomto případě se na databázi odešle pouze jediný dotaz, který bude vracet klienty i s objednávkami zároveň. Zatímco lazy loading načte pouze klienty a na jejich objednávky se dotazuje až na vyžádání separátními dotazy, eager loading automaticky načte všechna data najednou.

Transakce

Entity Framework nabízí uživateli metodu `BeginTransaction()`, která mu dává kontrolu nad začátkem a ukončením transakce v rámci existujícího kontextu. Další možností je použití `UseTransaction()` dovolující kontextu využít transakci, jež byla započata mimo Entity Framework.

```
using (var transaction = db.Database.BeginTransaction(System.Data.IsolationLevel.Serializable))
{
    try
    {
        var accountFrom = db.Accounts.Where(a => a.AccountId == idfrom).Single();
        accountFrom.Balance -= amount;
        var accountTo = db.Accounts.Where(a => a.AccountId == idto).Single();
        accountTo.Balance += amount;
        var mytransaction = new Transaction { TransactionFromId = idfrom, TransactionToId = idto,
            Amount = amount, Date = DateTime.Now };
        db.Transactions.Add(mytransaction);
        db.SaveChanges();
        transaction.Commit();
    }
    catch (Exception)
    {
        transaction.Rollback();
    }
}
```

Výpis 7: Entity Framework - Transakce

Ukázka z výpisu 4 dělá ve výsledku to stejné co Dapper v ukázce 1. Rozdíl je v tom, že nebylo nutné psát žádné vlastní metody ani SQL dotazy. Např. pro materializaci všech účtů v databázi stačí jednoduše příkaz `db.Accounts.ToList()`. Další podstatnou změnou proti Dapperu je sledování změn entit v kontextu. V této ukázce jsem se dotázal na příslušné účty z databáze, vytvořil jejich objekty v paměti, změnil některé jejich vlastnosti a nakonec je uložil zpět do databáze. Entity Framework standardně automaticky sleduje změny entit, proto ví, že daným účtům byl změněn jejich zůstatek. Při volání metody `SaveChanges()` tak automaticky na databázi odešle `UPDATE` dotazy, čímž permanentně změny uloží. V případě vložení nové transakce se pochopitelně jedná o dosud neexistující záznam, tudíž v tomto případě bude na databázi odeslán příkaz `INSERT`.

Nepovinný parametr `Isolation.Level.Serializable` jsem uvedl pouze pro demonstrační účely. V případě jeho vynechání použije Entity Framework úroveň izolace shodnou s nastavením databáze, čož je např. v případě MSSQL Serveru nastavení read committed. [6]

Dotazovací API

V současné době EF 6.1 poskytuje tyto možnosti, jakými lze získat data z databáze:

- LINQ to Entities - konvertuje LINQ dotazy na posloupnost příkazů, které posílá na Entity Framework a vrací objekty, jež mohou být použity EF i LINQem.

- Entity SQL - nezávislý na DB, podobný SQL. Je přímo zpracováván Object services a vrací ObjectQuery.
- Nativní SQL - vlastní SQL lze předat jako parametr pro metodu `SqlQuery()`.

ID generátory

1. Identity - identity sloupce pro MSSQL Server, MySQL, DB2 a Sybase
2. GUID - globálně unikátní ID pro objekty, programy, záznamy apod. Jedná se o 16 bytový binární datový typ.
3. Assigned - identifikátor přidělen aplikací

2.2.3 Entity Framework - shrnutí

Tato knihovna byla ještě do nedávna uzavřeným projektem. V říjnu 2013 přesla s verzí 6 na open source a od té doby ušla dlouhou cestu. Jen výpis zásadnějších novinek pro EF6 je poměrně rozsáhlý:

- Transakce - zlepšení práce s transakcemi + podpra externích transakcí.
- Asynchronní dotazy - podpora asynchronního programování s využitím async/await příkazů pro platformu .NET 4.5.
- Logování a odchytávání databázových operací - každý příkaz odeslaný frameworkem na databázi je možno zachytit (a případně upravit) aplikačním kódem.
- Testování s in-memory datasetem - možnost testování pouze v paměti prostřednictvím kontextu, jehož chování je definováno uživatelskými testy.
- Connection resiliency - automatické opakování příkazů, které selhaly z důvodu výpadku spojení.
- Connection management - spojení předávané jako parametr konstruktoru už nemusí být uzavřené. Dále je ponecháno více volnosti kódu a spojení se ukončuje až se zrušením kontextu (volání `Dispose()`).

Další důležité aspekty jako podpora uložených procedur nebo Linq to entities byly přítomny už ve starších verzích.

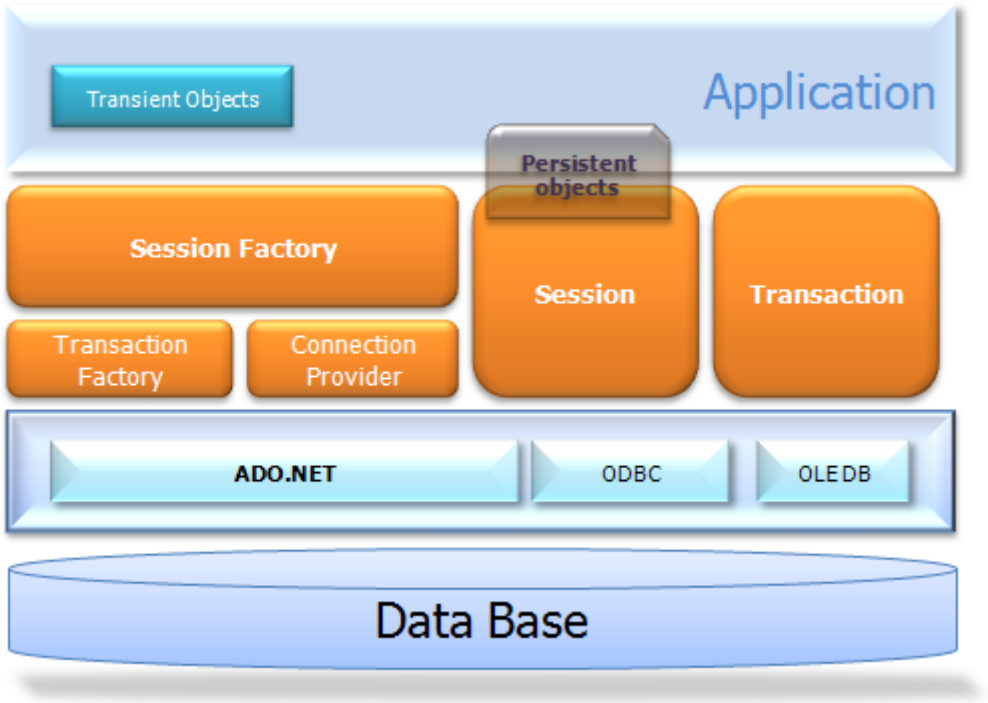
Dále si Entity Framework zvolil Microsoft jako svoje vlastní ORM řešení pro ADO.NET. Na jeho vývoji kromě různých členů komunity pracuje zkušený tým developerů, přičemž podle dat ze stránek `http://www.ohloh.net/p/entityframework` můžeme říci, že poměrně aktivně. Jedná se tedy o stabilní dlouhodobý projekt s dobrými vyhlídkami do budoucna.

Vzhledem k podporovaným technologiím je zřejmé, že tento projekt je cílen spíše do enterprise sféry. Nemohu nezmínit přehlednou webovou prezentaci s výborně zpracovanou a dobře provázanou dokumentací včetně videotutoriálů. Typické využití Entity Frameworku si lze představit např. v aplikacích využívajících WCF Data services (používá stejný Entity Data Model jako EF) nebo ASP.NET.

2.3 NHibernate

NHibernate je další z řady open source ORM frameworků pro .NET. Jedná se o port ORM s názvem Hibernate, který funguje na Javě. Podobně jako Entity Framework, i NHibernate ulehčuje programátorům práci tím, že usnadňuje přechod přes bariéru mezi objektovými a relačními daty. Je to "kompletní"ORM, svými možnostmi se tedy více podobá Entity Frameworku - mapuje .NET třídy na databázové tabulky, poskytuje nástroje pro dotazování a získávání dat, generuje SQL atd. [7]

Architektura Interně NHibernate pracuje takto:



Transient Objects: Instance tříd, které právě nejsou asociovány s žádnou session. Mohly být vytvořeny buď aplikací, nebo uzavřenou session.

Persistent objects: Krátkodobě žijící objekty obsahující status perzistence. Mohou to být POCO objekty, ale v dané době jsou asociovány právě s jednou session. Po jejím ukončení je bude možné použít v jakékoliv aplikační vrstvě.

Session Factory: Threadsafe (vzor immutable) cache zkompilovaných mapování pro jednu databázi. Vytváří session a connection provider. Může obsahovat i volitelnou cache druhé úrovně.

Session: Jednovláknový objekt s krátkou životností, který představuje komunikaci mezi aplikací a databází. Zapouzdřuje ADO.NET spojení a vytváří transakce. Také obsahuje cache první úrovně perzistentních objektů k jejich procházení/vyhledávání.

Transaction: Volitelný objekt s omezenou životností, jehož aplikace používá k zajištění atomických jednotek práce. Abstrahuje aplikaci od ADO.NET transakce. Jedna session může obsahovat více transakcí.

Transaction factory: Vytváří instance jednotlivých transakcí. Aplikace o ní neví, ale může být rozšířena vývojáři.

Connection provider: Vytváří ADO.NET spojení o příkazy. Abstrahuje aplikaci od konkrétní implementace IDbConnection a IDbCommand výrobcem.

2.3.1 Instalace

Podobně jako skoro se vším, i NHibernate nejpohodlněji nainstalujeme přes NuGet. Alternativně jej můžeme stáhnout z jeho domovské stránky - http://nhforge.org/. Pokud jste se přece jen rozhodli stáhnout zip soubor z webobých stránek, stačí jej rozbalit někam na disk a pro daný projekt ve Visual Studiu přidat reference na NHibernate.dll a NUnit.dll.

2.3.2 Technologie

Před použitím nainstalovaného NHibernate je potřeba provést jisté kroky k zajištění jeho funkčnosti. V prvé řadě musí být vytvořen konfigurační soubor, který definuje chování NHibernate jako spojení a dialekt databázového serveru, nastavení výpisu odesílaného SQL, apod. Tento soubor má přesně daný název - hibernate.cfg.xml. Vypadá nějak takto:

```
<?xml version="1.0" encoding="utf−8"?>
<hibernate−configuration xmlns="urn:nhibernate−configuration−2.2" >
  <session−factory>
    <property name="connection.driver_class">NHibernate.Driver.SqlClientDriver</property>
    <property name="connection.connection_string">
      Server=.\BP; initial  catalog=Fabrics;Integrated Security=SSPI
    </property>
    <property name="dialect">NHibernate.Dialect.MsSql2008Dialect</property>
    <property name="show_sql">false</property>
    <mapping assembly="NH_lazy"/>
  </session−factory>
</hibernate−configuration>
```

Výpis 8: Konfigurační soubor NHibernate

V balíku NHibernate jinak existují konfigurační šablony pro další databázové providery. Atribut `connection.driver_class` určuje typ databáze, se kterým se bude NHibernate spojovat - Oracle, Microsoft, atd. `connection.connection_string` pak představuje spojení na konkrétní server a databázi, dialect představuje konkrétní verzi

severu. Konfigurace ve výpisu 8 NHibernate říká, že se má spojit s MSSQL Serverem 2008, přesněji instancí serveru nesoucí název BP, na níž se nachází databáze se jménem Fabrics. Parametr `show_sql`nastavuje zapnutí/vypnutí zobrazování SQL kódu, který NHibernate vytváří. Stejným způsobem se pomocí hbm.xml souborů popíší i mappery (jejich build action je však potřeba nastavit na Embedded resource), doménové entity lze zapsat formou klasických POCO tříd.

Alternativou k těmto XML souborům jsou tzv. **Code-based konfigurace**. Jedná se o nastavení NHibernate skrze striktně kontrolovaný `C#` kód, což poskytuje výhody v podobě typové kontroly, snadného refaktoringu atd. Zdaleka nejpopulárnější nástroj pro tyto konfigurace je rozšíření nesoucí název Fluent NHibernate, popřípadě je možno využít i integrované API mapping-by-code.

Lazy loading

Tuto strategii načítání objektů využívá NHibernate implicitně, nicméně pro její korektní funkčnost musí být property v POCO třídách definovány jako `virtual`, navíc se v mapperech nesmí vyskytovat nastavení `lazy=false`.

```
var client = session.Get<Client>(idclient);
Console.WriteLine("Vsechny objednavky pro Id: {0}, {1} {2}", client.ClientId, client.FirstName,
    client.LastName);
foreach (var item in client.Order)
{
    Console.WriteLine("{0} {1} {2} {3} {4}", item.OrderId, item.ClientId, item.OrderDate, item.
        OrderTotal, item.OrderStatus);
}
```

Výpis 9: NHibernate - Lazy loading

Tato ukázka je stejný příklad jako výpis 5 v Entity Frameworku - získání dat o klientovi a všech jeho objednávkách. Vzhledem k dotazovacím možnostem NHibernate existuje více způsobů, jak daný úkon řešit. Ukázka značí použití metody `Get<T>`, která poskytuje způsob, jak získat entity na základě primárního klíče. Výhodou je, že tato metoda automaticky nejdříve kontroluje session cache i cache druhé úrovně. Výsledkem budou opět dva odeslané SQL dotazy - jeden pro informace ohledně klienta, druhý pro jeho objednávky. NHibernate pochopitelně také podporuje **eager loading**, o jehož vynucení se stará metoda `Fetch()`.

Dotazovací API
NHibernate poskytuje hned několik způsobů, jakými lze z databáze získat data:

```
session.CreateCriteria(typeof(Client)).Add(Restrictions.Eq("ClientId", idclient)).CreateCriteria("
    Order").UniqueResult<Client>(); -> Criteria API

session.Query<Client>().Where(c => c.ClientId == idclient).Fetch(c => c.Order).ToList(); -> LINQ

session.CreateQuery("from Client as c where c.ClientId = :idclient").SetParameter("idclient",
    idclient).List<Client>(); -> HQL
```

Výpis 10: NHibernate - dotazy

- Criteria API - představuje dotaz na danou perzistentní třídu. Umožňuje programově vytvářet dotazy, vhodné pro dynamické dotazování.
- HQL - podobné SQL, nicméně plně objektově-orientované s podporou dědičnosti, polymorfismu a asociací.
- LINQ - integrován od NHibernate verze 3. Možno využít metodu `Query()`.
- Nativní SQL - vlastní SQL lze předat jako parametr pro metodu `CreateSQLQuery()`.

Transakce
NHibernate nativně podporuje transakce a je důrazně doporučeno je používat v podstatě neustále, tedy i např. pro read operace.

```
using (ISession session = NHibernateHelper.OpenSession())
{
    using (ITransaction transaction = session.BeginTransaction())
    {
        var accFrom = session.CreateCriteria(typeof(Accounts)).Add(Restrictions.Eq("AccountId",
            idfrom)).UniqueResult<Accounts>();
        accFrom.Balance -= amount;
        var accTo = session.CreateCriteria(typeof(Accounts)).Add(Restrictions.Eq("AccountId", idto))
            .UniqueResult<Accounts>();
        accTo.Balance += amount;
        session.Update(accFrom);
        session.Update(accTo);
        session.Save(helper.AddTransaction(idfrom, idto, amount, accFrom, accTo));
        transaction.Commit();
    }
}
```

Výpis 11: NHibernate - Transakce

Důvodem budiž skutečnost, že každá databázová operace je prováděna v rámci transakce (včetně čtení). Pokud nedefinujeme konkrétní transakci, bude použit implicitní režim, kde se každý příkaz na databázi provede ve své vlastní transakci, což má za následek sníženou výkonnost a konzistenci. Dalším důvodem je 2nd level cache (cache druhé

úrovně). NHibernate se snaží, aby si tato cache udržovala konzistentní obraz databáze, čehož dosahuje tak, že se všechny její aktualizace odkládají až na commit transakce. Pokud se tedy transakce explicitně nedefinují, není nikdy zavolán `commit()`, tudíž NHibernate mezipaměť neaktualizuje daty načtených entit (jako by neexistovala). [8]

ID generátory
NHibernate podporuje celou škálu strategií, s jejichž pomocí lze vytvářet ID hodnoty primárních klíčů.

1. Increment - vytváří identifikátory typu Int16/32/64, které jsou ovšem unikátní pouze v případě, že žádný další proces nevkládá data do stejné tabulky.
2. Identity - identity sloupce pro MSSQL Server, MySQL, DB2 a Sybase
3. Sequence - sekvence pro Oracle, PostgreSQL, Firebird
4. Hilo - použití hi/lo algoritmu ke generování identifikátorů typu Int16/32/64 na zakládě sloupce a tabulky. Jsou unikátní jen pro danou databázi.
5. Seqhilo - stejný jako hilo, ale parametrem je pojmenovaná databázová sekvence.
6. UUID - vytvoří id pomocí 128bit UUID algoritmu. Navrácené id je typu string unikátní v rámci sítě (použití IP).
7. GUID - databází generovaný string pro MSSQL Server a MySQL.
8. Assigned - identifikátor objektu přiděluje aplikace před voláním metody Save. Implicitní volba.
9. Native - bude vybrán identity, sequence, nebo hilo podle možností dané databáze.
10. Foreign - použije identifikátor jiného asociovaného objektu. Většinou používaný spolu s 1:1 asociací.

2.3.3 NHibernate - shrnutí

Vznik této knihovny se datuje někam do roku 2005, kdy ji ještě zajišťovala firma JBoss. Přibližně o rok později byla tato podpora ukončena a od té doby je tento ORM veden a vyvíjen komunitou. K dnešnímu dni byl nasazen do tisíců projektů. [9]

NHibernate je tedy výrazně starší než Entity Framework nebo Dapper. Z toho plyne jeho značná vyzrálost a flexibilita, kterou můžeme vidět např. na jeho dotazovacích schopnostech, id generátorech, podporovaných databázích, 2nd level cache (různé možnosti), podpoře batchingu atd.

Na druhou stranu všechny tyto možnosti mohou být do jisté míry překážkou pro nové vývojáře - např. mají použít Criteria API, nebo HQL? V některých oblastech jde NHibernate sám proti sobě. Vzhledem k určitým nevýhodám mapování přes XML soubory (typová kontrola, refaktoring atd.) vzikl projekt třetí strany nesoucí název Fluent

NHibernate, který tyto nedostatky eliminoval pomocí odstranění XML souborů a definici mapování v `C#`kódu. Logickým krokem by bylo zahrnout populární Fluent NHibernate do hlavního kódu NHibernate, ale místo toho se vývojáři rozhodli vyvinout vlastní API mapping-by-code.

V současné době se NHibernate nachází ve verzi 3.3.3, která se datuje k 8.8.2013, takže uběhl necelý rok od poslední aktualizace. Situace není o mnoho lepší ani při pohledu na statistiky projektu na adrese `http://www.ohloh.net/p/nhibernate`, kde lze vypozorovat poměrně značný meziroční pokles aktivity. Vzhledem k této skutečnosti a také posilování Entity Frameworku bych se bál NHibernate doporučit jako platformu pro nový projekt.

2.4 Srovnání ORM knihoven

V této kapitole se pokusím jednotlivé ORM knihovny mezi sebou porovnat na základě jejich vlastností a schopností.

Dapper se od dalších dvou ORM v této práci poměrně liší. Jak už bylo napsáno výše, jedná se o tzv. micro-ORM, což znamená, že tato knihovna mimo jiné neřeší stavy a asociace mezi objekty, nemá žádné dotazovací API kromě ručně psaných SQL dotazů, nepodporuje lazy loading, ani se nestará o generování databázového schématu, případně jeho úpravu při změně kódu. Je to zkrátka pomocná knihovna, jejíž primárním cílem je mapovat .NET třídy na databázové tabulky a na základě výsledků SQL dotazů zpětně vytvářet objekty těchto tříd. Takto byl Dapper od začátku koncipován a navržen - dává přednost jednoduchosti a přímočarosti před plnou automatizací.

Od toho se odvíjí potenciální nasazení této knihovny. Jestliže daný projekt vyžaduje podporu např. migrací, odpojených entit, či interception, nebude Dapper vhodnou volbou z důvodu naprosté absence těchto technologií. Pokud je ovšem kladen důraz převážně na transfer dat mezi aplikací a databází, přičemž absence pokročilejších technologií nepředstavuje problém, pak bych Dapper doporučil pro jeho jednoduchost a prověřenost (nasazen na StackOverflow a dalších). Velké frameworky by v takovém případě mohly působit až zbytečně nafouknutě a vývojářům by zabralo delší dobu se s nimi seznámit.

Entity Framework a **NHibernate** jsou si v mnohých věcech podobné a přímo si konkurují. Pokusím se je porovnat na základě vlastností, které jsou zejména v enterprise sféře často využívané.

Databázová podpora: NHibernate by měl bez problému fungovat na všech známějších databázích - SQL Server, Oracle, Acces, Firebird, PostgreSQL, MySQL, SQLite. Entity Framework má momentálně podporu pro SQL Server, Firebird, Visual Fox Pro, MySQL a PostgreSQL. Pro Oracle či SQLite existují providery třetí strany, nicméně po představení EF6 bylo jeho code-first konfigurační API dostupné pouze pro SQL Server a na další providery bylo nutné čekat.

Dotazovací API: Criteria API, HQL, LINQ a nativní SQL dávají NHibernate značnou flexibilitu, nicméně vývojářům mohou připadat redundantní. Entity Framework nabízí LINQ to Entities, Entity SQL a nativní SQL.

Strategie generování ID: 3 základní u EF vs. 11 pro NH. I když generátory podporované EF jsou obecně nejpoužívanější, v ostatních případech má v tomto oledu NHibernate jednoznačně navrch.

Asynchronní dotazy: Entity Framework tuto technologii podporuje od verze 6. Implementace do NHibernate by vyžadovala značný refaktoring kódu a není plánována ani pro budoucí verzi 4.

Práce s odpojenými entitami: Typicky tato situace nastává např. ve webovém prostředí, kdy je kontext (obdoba session) uzavřen po zpracování požadavku a obsah entity je předán klientovi formou HTTP. Další HTTP požadavek poskytuje modifikovaný obsah entity, která musí být znovu vytvořena, přidána do nového kontextu a uložena. NHibernate dokáže v tomto scénáři bez problému uložit celý objektový graf, zatímco Entity Framework vyžaduje manuální synchronizaci každé entity, což je pro mnoho vývojářů nepřijatelné.

2nd level cache: NHibernate disponuje hned několika providery pro cache druhé úrovně, Entity Framework tuto technologii nepodporuje. Přinejlepším pro něj existuje rozšíření pro cacheování výsledků SQL dotazů, ale prozatím se nachází v beta stádiu. [10]

Migrace: Oba frameworky podporují migrace včetně těch automatických. EF má navíc Seed metodu vhodnou především k testování.

Lazy loading: NH podporuje tuto technologii pro asociované entity, kolekce a skalární property. EF pouze asociované entity a kolekce.

Code-based konfigurace: NH má své mapping-by-code, případně populárnější Fluent NHibernate ve formě rozšíření. EF má podporu už od verze 4.1.

Dokumentace: Entity Framework má kvalitně popsanou a aktuální dokumentaci včetně nejnovějších přídavků na jednom místě. Naproti tomu dokumentace NHibernate je neaktualizovaná, nové technologie v ní i dlouhou dobu po vydání chybí a informace je často nutné hledat rozházené různě na internetu.

Z výše popsaného je zřejmé, že každá knihovna má své výhody i nevýhody. Není možné jednoznačně říct, která je efektivnější. Je potřeba schopnosti každé knihovny promítnout do daného projektu a zjistit, zdali je potřebná technologie podporována, nebo ne.

3 Dokumentové knihovny

Dalším z nejčastěji používaných úložišť jsou dokumentové databáze známé též jako NoSQL databáze. Od relačních se liší tím, že namísto tabulek s vazbami ukládají data do dokumentů. Ty nemusejí mít stejnou strukturu pro každý záznam a mohou být takřka neomezeně komplexní. Výhodo těchto databází je vysoká dostupnost (dokumenty se jednoduše přenášejí mezi více servery) a modifikovatelnost (nevyžadují nějaké konkrétní schéma). Pochopitelně dokumenty v databázi jsou stále něco jiného než objekty v rámci běhu programu, takže podobně jako u ORM knihoven, i zde musíme zajistit přechod mezi objektovými a dokumentovými daty. [11]

Dokumentové databáze jsou od relačních odlišné také v tom, že nemají definovaný žádný standard, který by přesně říkal co a jak. Každá knihovna disponuje svým vlastním serverem a API pro vývoj.

3.1 RavenDB

RavenDB je open-source (pouze pro nekomerční využití) dokumentová (NoSQL) databáze napsaná v .NET (tedy nejen Client API, ale i databázový engine). Data jsou ukládána bez schématu jako JSON dokumenty, což je výhodné pro jednoduchou úpravu. RavenDB se skládá ze serveru a klienta. Server řeší ukládání dat a zpracování dotazů, které na něj klient zasílá. [12]

3.1.1 Instalace

RavenDB umí pracovat buď v klasickém klient-server, nebo embedded módu, kdy se spouští jako proces zároveň s aplikací. Aplikace v přiložené solutione je psaná pro klient-server mód, takže bude nutné nainstalovat RavenDB Client i RavenDB server - oba je možné získat přes NuGet. Alternativně lze RavenDB stáhnout z jeho oficiálních stránek v pravém menu pod Latest Downloads - http://ravendb.net/download. Server se spouští souborem Raven.Server.exe (NuGet jej zpravidla nahraje do složky packages) a při spuštění projektu pochopitelně musí běžet. Dále je možné server v případě potřeby nainstalovat jako službu, případně jej integrovat do IIS na Windows systémech.

3.1.2 Technologie

RavenDB je tzv. schema-less databáze, což znamená, že typicky narozdíl od relačních databází nemá žádné schéma, které by pevně definovalo její strukturu. Nevyžaduje proto žádné speciální mapování, postačí pouhá definice skrze POCO entity. Jednotlivé dokumenty jsou pak ukládány ve formě JSON dokumenů.

Management Studio

Každou instanci RavenDB serveru je možno spravovat pomocí této vzdáleně přístupné Silverlight aplikace. Pro tento účel stačí do některého webového prohlížeče zadat adresu serveru a port, na kterém naslouchá (na daném počítači zpravidla localhost:8080).

Studio slouží jako vzdálený správce dané instance RavenDB serveru, umožňuje tedy provádět operace jako např. CRUD na dokumentech, správu indexů, psaní dotazů, zobrazení logů, import/export atd.

Transakce

Pro zajištění transakcí RavenDB využívá implementaci Microsoftu - třídu TransactionScope a technologii DTC, které je možné využít nejen pro vícero operací na jeden server, ale také na více serverů a databází (distribuované transakce).

```
using (IDocumentSession session = documentStore.OpenSession())
{
    using (var transaction = new TransactionScope())
    {
        try
        {
            User user = session.Load<User>("users/"+userid);
            user.Name = username;
            session.SaveChanges();
            transaction.Complete();
            Console.WriteLine("Vsechny operace uspesne dokonceny.");
        }
        catch (Exception)
        {
            Console.WriteLine("Chyba, rollback.");
        }
    }
}
```

Výpis 12: RavenDB - Transakce

Výňatek z metody ve výpisu 12 mění jméno uživatele dle zadaných parametrů. Jako v každé transakci i zde buď proběhnou všechny operace úspěšně a změna se zapíše do databáze, nebo některá selže a databáze zůstane v původním stavu.

Oproti relačním databázím si lze všimnout změny při načítání uživatele (metoda `Load()`). Místo konvertování vstupu na integer a jeho odeslání jakožto parametru se na databázi pošle ID ve tvaru např. `users/1`. Je to z toho důvodu, že RavenDB při vynechání ID pole danému dokumentu automaticky vygeneruje jeho ID, které je standardně ve formátu <entita>/číslo, tedy např. `blogs/1` (názvy entit jsou pluralizovány). Pochopitelně se jedná spíše o pouhé konvence a není nutné je dodržovat. Reálně může ID být jakýkoliv string.

Z ukázky je také patrné, že jednotkou práce (unit of work) RavenDB je IDocumentSession, jež je vytvářena v rámci IDocumentStore (továrna). Podobně jako např. u NHibernate, DocumentSession komunikuje s databází (přenos dat, dotazy) a jedná se o lehký objekt, zatímco DocumentStore řídí klient/server komunikaci, vyžaduje více zdrojů a měla by se vytvářet pouze jednou pro celou aplikaci. [12]

Replikace
Je proces kopírování a spravování databázových objektů mezi více databází, které dohromady tvoří distribuovaný databázový systém, což může zvýšit výkonost a zlepšit dostupnost dat. Zapnutí replikačního modulu bude mít za následek:

- Zaznamenávání serveru, na který byl dokument původně zapsán. Slouží pro kontrolu konfliktu mezi replikovaným a původním dokumentem.
- Konfliktní dokumenty budou označeny a bude zapotřebí automatické nebo uživatelské reakce.
- Mazané dokumenty obdrží tzv. delete markers (označení pro smazání). Ty slouží k replikování jednotlivých mazání na potomcích.
- Replikace nefunguje pro systémové dokumenty, jejichž klíč začíná Raven/.

K nastavení replikace je dále nutné vytvořit dokument Raven/Replication/Destinations, což je seznam serverů, na které se má replikovat. Samotná replikace pak probíhá tak, že při každém úspěšném zakončení transakce (commit) se Raven podívá na seznam destinací replikace, přičemž pro každou destinaci se dotáže na poslední dokument, jež zde byl replikován. Následně zašle dávky updatů, které nastaly od poslední replikace. To vše probíhá paralelně na pozadí. [12]

Lazy loading
Raven podporuje lazy operace pro dotazování, načítání, filtrované vyhledávání a našeptávání. K použití této technologie se využívá metoda `Lazily()`, která označí jakýkoliv typ dotazu jako lazy operaci.

```
Lazy<IEnumerable<User>> lazyUsers = session.Query<User>().Lazily();
IEnumerable<User> users = lazyUsers.Value;
```

Výpis 13: RavenDB - Lazy loading

Takto označený dotaz vrátí lazy instanci, která bude vyhodnocena až v případě potřeby, což v této ukázce provádí příkaz Value.

Sharding
Jedná se o technologii, která poskytuje způsob, jak rozdělit data na více serverů tak, že každý server obsahuje jen jejich určitou část. To je výhodné hlavně v případě, že databáze obsahuje velké množství dokumentů (spoustu indexů), jelikož se tak rozloží zátěž mezi více serverů. Raven řeší všechny aspekty shardingu, takže uživateli zbývá specifikovat, jak rozdělit dokumenty mezi vícero shardů. K zajištění shard funkcionality je potřeba místo DocumentStore použít ShardedDocumentStore, který se ale jinak až na inicializační fázi chová stejně. Dále je nutné specifikovat ShardStrategy, jež obsahuje slovník se shardy, na kterých má pracovat - tato instance se předává ShardedDocumentStore při jeho inicializaci. Slovník obsahuje ID shardu a DocumentStore instance, která na něj odkazuje.

Vyhledávání

Poměrně často používaná funkce. Raven nabízí několik možností, jak použít určitou formu vyhledávání v dokumentech:

```
foreach (var user in Queryable.Where(session.Query<User>(), x => x.Name.StartsWith(
    searchname)))

var users = session.Query<User>("UsersByName").Search(x => x.Name, searchname + "␣" +
    searchname2).ToList();
```

Výpis 14: RavenDB - Vyhledávání

Výpis 14 obsahuje dva příklady. První řádek vyhledává na začátku jména (např. ma -> Martin), druhý pak hledá fráze s přesnou shodou (tedy všechny znaky v textovém poli se musí shodovat). Další možností je vytvoření složeného indexu, díky kterému je pak možné vyhledávat na dvou nebo více property zároveň (např. jméno a povolání zároveň).

Cache

Raven má vlastní implementaci vyrovnávací paměti. Továrně je tato paměť zapnutá a je možno nastavit jak cacheování pro danou URL, tak i počet takto uložených požadavků. Dále lze cache nastavit do tzv. agresivního módu. V případě použití tohoto nastavení se Raven ani neptá serveru, ale rovnou vrátí odpověď z lokální cache (pokud tam je), což je pochopitelně výrazně rychlejší. Implicitně klient naslouchá notifikacím ze serveru a s jejich využitím je schopný anulovat dokumenty v cache, u kterých proběhla změna. Takto ví, kdy se musí zeptat serveru a kdy může dokumenty načíst z cache, nicméně je možné získat špatná data, jelikož obdržení notifikací ze serveru zabere nějaký čas. [12]

3.1.3 RavenDB - shrnutí

RavenDB poprvé spatřil světlo světa zhruba polovině roku 2010 a jeho současná poslední verze 2.5.2879 pak 11.5.2014. Je zdarma v rámci OSS, komerční projekty si musejí zakoupit některou z nabízených licencí.

Tato dokumentová databáze je kompletně vyvíjena na .NET platformě. Klade velký důraz na bezpečnost a spolehlivost dat (ACID transakce, implicitní limity na počet přenesených dokumentů, batching apod.) a enterprise sféru obecně s využitím technologií jako replikace či sharding (oba lze použít zároveň). Data jsou v Ravenu ukládána ve formě JSON dokumentů bez schématu, tudíž není potřeba se zatěžovat jeho případnými změnami nebo tvorbou specifického mapování. Primárním nasazením by dle tvůrců mělo být webové prostředí. Tato knihovna může operovat v klasickém klient/server nebo embedded módu, kdy je přímo spjata s danou aplikací. Také existují rozšíření ve formě balíčků, jež propůjčují dodatečnou funkcionalitu - např. komprese, šifrování, verzování, nebo již zmíněnou replikaci a sharding.

V řádu několika týdnů by měla vyjít nová verze - RavenDB 3.0, která představuje některé zajímavé novinky - mimo jiné zlepšení výkonnosti a flexibility indexace, RavenFS (replikovaný file systém s podporou velkých souborů a jejich správy), JVM API, nebo Voron, což je nový storage systém, jenž nahraní současný Esent.

3.2 Redis

Redis je open source NoSQL databáze vyvíjená za podpory společnosti Pivotal. Nejedná přímo o dokumentovou databázi jako RavenDB, ale o tzv. key-value store (pracuje na principu klíč -> hodnota). Hlavní rozdíl proti dokumentové databázi spočívá v tom, že key-value store lze dotazovat pouze na hodnotu klíče, zatímco u Dokumentové databáze se můžeme ptát na jakýkoliv atribut, jelikož se záznam skládá z více položek. Key-value store je tedy jednodušší a často rychlejší na úkor flexibility.

Redis je obvzláště zajímavý jednou věcí - s celou databází pracuje kompletně v paměti. Díky tomu (a faktu, že se jedná o key-value store) dosahuje extrémních výkonů. Pochopitelně z tohoto přístupu plynou jisté nevýhody - z důvodu in-memory operací je zapotřebí, aby se celý dataset vešel do paměti, což v některých případech může představovat problém. Dále je na místě otázka, jak Redis zajišťuje persistenci dat, když se RAM paměť po restartu/vypnutí maže? Tento problém však Redis řeší pomocí persistenčních technologií RDB a AOF. Více se o nich dočtete níže v dokumentu.

Redis byl původně vyvíjen pouze pro Linux, nicméně práce probíhajíi také na verzi pro Windows pod záštitou Microsoft Open Tech skupiny. V současné době již tento port Redisu dosáhl produkční verze.

3.2.1 Instalace

V rámci této práce jsem s Redisem pracoval ve Windows prostřednictvím .NET klienta. K dnešnímu dni je možné Redis server i `C#` klient získat přes NuGet. Stačí vyhledat klíčové slovo Redis a stáhnout první nalezenou položku `C#` Redis client s Id ServiceStack.Redis. Redis server doporučuji stáhnout z GitHubu - https://github.com/MSOpenTech/redis (tlačítko Download ZIP napravo). Obsah balíčku postačí rozbalit někam na disk a ve složce msvs otevřít soubor redisserver.sln ve Visual Studiu (alespoň 2013). Pak stačí už jen spustit kompilaci, případně nastavit build konfiguraci či platformu. Po kompilaci lze Redis server zapnout souborem redis-server.exe.

3.2.2 Technologie

Podobně jako Raven, ani Redis nevyžaduje nějaké konrétní schéma. Bohatě mu postačí definice modelu skrze POCO entity. Tyto entity jsou pak serializovány ve formě JSON.

Perzistence

V úvodu sekce jsem zmínil dva způsoby, jakými Redis zajišťuje persisnteci dat - RDB a AOF.

RDB pracuje tak, že v nastavených intervalech pořizuje snapshoty databáze což s sebou přináší určitě výhody a nevýhody. Mezi výhody patří, že se jedná o reprezentaci databáze v určitém čase v jednom malém souboru, což je výhodné např. pro přenos tohoto souboru do vzdálených datacenter v případě chyby. Dále RDB umožňuje Redis serveru dosáhnout maximálního výkonu, jelikož hlavní proces serveru nemusí dělat žádné diskové operace, pouze vytvoří podproces, který se o vše postará. Mezi nevýhody patří ztráta dat od posledního snapshotu v případě problémů (např. výpadek proudu) nebo

fakt, že pro jakékoliv diskové operace musí server vytvořit podproces, což může zabrat dost procesorového času v případě velkého datasetu a zpomalit tak komunikaci serveru s klienty. [13]

AOF (append-only file) naproti tomu zaznamenává každou write operaci, která bude přehrána při dalším spuštění serveru a ve výsledku tak databázi obnoví. Z toho důvodu má AOF lepší možnosti nastavení synchronizace (každou vteřinu, pro každý dotaz). Když soubor začne být příliš velký, umí jej server přepsat tak, že vytvoří nový soubor s minimálním množstvím operací potřebných k vytvoření databáze. Server zapisuje změny do starého souboru a jakmile je nový soubor připraven, začne zapisovat do něj. Mezi nevýhody patří větší velikost AOF souborů oproti RDB souborům a podle nastavení synchronizace může být AOF pomalejší, i když při nastavení jedné vteřiny je výkon pořád vysoký. [13]

Redis dovoluje použít i kombinaci RDB a AOF. Také je možné perzistenci úplně vypnout; v takovém případě budou data v paměti pouze po dobu běhu serveru a zaniknou s jeho vypnutím.

Transakce

Narozdíl od Ravenu nebo dalších knihoven, transakce v Redisu fungují poněkud odlišně, i když deklarace může na pohled vypadat podobně.

```
var users = redisclient.As<User>();
var user = redisclient.GetById<User>(userid);
using (var transaction = client.CreateTransaction()) //MULTI
{
    user.Name = username;
    transaction.QueueCommand(x => x.Store(user));
    transaction.Commit(); //EXEC
} //DISCARD, pokud nebyl vyvolan EXEC
```

Výpis 15: Redis transakce

Redis k jejich kontrole používá klíčová slova multi, exec, discard a watch, přičemž všechny transakce musejí splňovat určité podmínky: všechny příkazy jsou v rámci transakce serializovány a zpracovány sekvenčně. Nemůže se tedy stát, že žádost jiného klienta je vykonávána uprostřed průběhu některé transakce, díky čemuž jsou příkazy vykonány jako jediná operace. Druhou podmínkou je, že se buď provedou všechny příkazy, anebo žádný; Redis transakce je tedy atomická. Příkaz exec spustí provádění všech příkazů v transakci, takže pokud klient ztratí spojejí se serverem ještě před voláním multi, žádná z operací se neprovede, v opačném případě se provedou všechny. Watch lze použít ke sledování klíče - spuštění exec podmíní tím, že žádný další klient nezměnil některý ze sledovaných klíčů.

Příkazy jako exec, multi a další se v kódu ukázky nevyskytují (znázorněno komenty), jelikož ServiceStack `C#` klient pro Redis používá jinou syntaxi, nicméně z hlediska funkčnosti zde není rozdíl.

Další zajímavostí transakcí v Redisu je, že nepodporují rollback. V transakci se zpravidla vyskytne jedna ze dvou chyb: příkaz může mít chybnou syntaxi (např. špatný počet argumentů, případně i plná paměť) a chyba nastane ještě před voláním exec. Druhá

chyba se vyskytne až po volání exec (např. použití příkazu na špatný klíč - string -> int apod.). Ve starších verzích Redisu tak mohly vznikout situace, kdy se do databáze dostaly pouze příkazy, které bez problémů prošly kontrolou a byly přidány do fronty (metoda `QueueCommand()`). Od verze 2.6.5 si však server pamatuje, že nastala chyba při řazení příkazů do fronty, a tak transakci vůbec neprovede a zahodí ji.

Redis také nedovoluje načtení a manipulaci s daty v rámci jedné transakce, což je důvod, proč se v ukázce 15 uživatel načítá před jejím začátkem.

Datové typy

Redis je sice koncipován jako key-value store, ale narozdíl od typického key-value storu, který spojuje string klíče se string hodnotami, Redis podporuje různé typy dat:

- **Keys** - jsou binárně bezpečné, takže lze jako klíč použít jakoukoliv binární sekvenci.
- **Lists** - listy stringů. Je možné přidávat elementy na jejich začátek nebo konec.
- **Sets** - neuspořádané kolekce stringů. Nedovolí opakování stejných členů (v takové situaci bude set obsahovat pouze jeden člen).
- **Hashes** - skupina polí, kde je každé asociováno s nějakou hodnotou. Vhodné pro ukládání objektů.
- **Sorted sets** - neopakující se kolekce stringů. Narozdíl os setu je každý člen asociován se skóre, nebo podle hodnosti. Umožňují rychlý přístup ke všem členům.

Replikace

Redis používá asynchronní master-slave replikaci, která umožní slave serverům, aby byly přesnými kopiemi master serverů. V praxi si master ukládá logy ohledně databázových operací, ty následně posílá slavům, kteří vykonávají příkazy tak, aby měli ve výsledku stejný dataset. Jeden master tedy může mít více slavů, přičemž slave servery mohou akceptovat spojení od dalších slavů. Při spojení vyšle slave synchronizační příkaz, načež master začne ukládat na pozadí a bufferovat všechny příkazy, jež modifikují dataset. Jamile tento proces dokončí, přenese master databázový soubor na slave, který si jej uloží na disk a načte do paměti. Nakonec master přenese buffer, což se děje ve formě streamovaných příkazů. [13]

Sharding

Neboli rozdělení dat podle klíče na různé servery. Pro Redis je tato technologie významná hlavně ze dvou důvodů: jelikož je k dispozici více počítačů, mají dohromady efektivně více paměti, což dovoluje nasazení větší databáze. Druhým důvodem je velmi dobré škálování výkonu, protože se sčítají prostředky každého z počítačů, což kromě procesoru platí i pro síťovou propustnost. Redis nabízí více způsobů shardingu, mimo jiné např. range (podle rozsahu ID) nebo hash (na jakýkoliv klíč se uplatní hash funkce, která jej přemění na číslo).

Mezi nevýhody patří např. nemožnost použití transakcí operujících na více klíčích, větší komplexita dat (více RDB/AOF souborů), nebo přidávání/odebírání uzlů za běhu, které ostatní systémy nepodporují.

3.2.3 Redis - shrnutí

Redis (Remote dictionary server) je pokročilý key-value store, který byl původně vypuštěn 10.4.2009. Dokáže ukládat data nejven ve formě stringů, ale také listů, setů, nebo hashů. Jedná se o kompletně open-source software distribuovaný pod BSD licencí.

Jeho specifickou vlastností je, že s celým datasetem operuje pouze v paměti, díky čemuž může dosahovat extrémních výkonů. Z toho důvodu se jej rozhodlo nasadit mnoho projektů, ať už jako hlavní databázi, nebo formou pomocné cache. Aktuálně lze Redis najít na Twitteru, Githubu, Flickeru, StackOverflow a dalších. Jeho popularita vedla ke vzniku klientů snad v každém známějším jazyce - C/C#/C++, Java, Python, PHP atd.

Z pohledu enterprise sféry je Redis zajímavý především svou výkonností, ale také velmi dobrou škálovatelností skrze technologie jako replikace a sharding. Omezení maximální velikosti databáze na velikost operační paměti je poměrně nepříjemné, nicméně je na každém, aby zvážil, jak velkou prioritu pro něj má rychlost a s jak velkým datasetem bude pracovat.

3.3 Srovnání NoSQL knihoven

Zde se opět pokusím provést srovnání, tentokrát mezi dokumentovými databázemi - RavenDB a Redis.

Databázový model: Raven je dokumentová databáze, která nevyžaduje žádné schéma a využívá Esent jako storage engine. Kromě klient-server může pracovat i v embedded módu uvnitř aplikace. Redis je pokročilý key-value store, také nevyžaduje žádné schéma, nicméně celá databáze se musí vejít do paměti, čímž dává přednost rychlosti na úkor flexibility.

Dotazovací API: Raven umožňuje dotazování přes LINQ nebo HTTP RESTful API, které jsou překládány na Lucene dotazy a vykonány vůči vhodnému indexu. Redis z principu key-value store žádné specifické dotazování neumožňuje, pouze má hodnoty asociované s klíči.

Škálovatelnost: Oba podporují master-slave replikaci i sharding.

Indexování: Raven používá dynamické (automatické) indexy, navíc uživateli umožňuje vytvořit si vlastní (statický) index. Také podporuje kompozitní klíče a full text search. Redis indexuje primární klíče, přičemž předpokládá, že každá entita má právě jeden primární klíč (nepodporuje kompozitní klíče). Full text search lze implementovat např. s využitím Lua s enkódováním/dekódováním JSON nad Redisem, nicméně továrně Redis tuto funkci nemá.

Platformy: RavenDB Server v současné době běží pouze pod Windows na .NET Framework 4.0, klient může běžet i na Monu. Redis je multiplatformní - funguje na Linuxu, BSD, OS X. Windows port je pak vyvíjen za podpory Microsoftu.

Vývojové API: Jen a pouze C# pro Raven, zatímco Redis má klienty pro všechny známější API - C, C#, C++, Clojure, Common Lisp, D, Dart, Erlang, Haskell, Java, Lua, Perl, PHP, Python, Ruby, Scala, Smalltalk a další.

Dokumentace: Obě knihovny mají na svých domovských stránkách obsáhlou a dobře popsanou dokumentaci. Nicméně v případě Redisu se stává, že konkrétní implementace

v rámci daného klienta se od dokumentace liší (např. nepřítomnost multi a exec v `C#` klientovi), pak je potřeba hledat jinde.

Licence: RavenDB je zdarma pro open-source projekty; v případě komerčního využití je nutné zakoupit licensi, jejíž cena se podle konfigurace pohybuje v rozmezí stovek až tisíců dollarů ročně. Redis je kompletně open-source projekt, tedy naprosto zdarma.

Faktem je, že RavenDB a Redis se od sebe poměrně zásadně liší už ve způsobu, jakým pracují - dokumentová databáze s rozsáhlými možnostmi dotazování a indexování versus in-memory key-value store.

Redis bude vhodnější všude tam, kde jsou kladeny náročné požadavky na výkon, ale zároveň není dataset tak velký, aby se nevešel do paměti serveru/serverů. V opačném případě jej však lze použít jako pomocnou cache. Pro zajištění perzistene Redis používá techniky RDB (snapshoty databáze) a AOF (záznam sekvence write operací), jež svůj účel splňují, nicméně se může stát, že v případě pádu se ztratí změny provedené za určitou jednotku času dle nastavení (např. interval mezi dvěma snapshoty apod.). Dále Redis trpí menší flexibilitou dotazování - umožňuje se dotazovat pouze na daný klíč a nelze mu např. říct, ať vrátí záznamy, jejichž pole věk je rovno 30.

RavenDB je tvůrci zamýšlen převážně do webového prostředí a OLTP systémů. Je schopen ukládat velice komplexní objekty, respektive objektové grafy, přičemž poskytuje intuitivní API, prostřednictvím kterého se lze dotázak na jakýkoliv atribut. Další přidanou hodnotu představují velice bezpečné ACID transakce, lazy loading nebo možnost spravovat databázi v grafickém režimu skrze webový prohlížeč.

Podobně jako v případě ORM frameworků, nelze jednoznačně doporučit tu či onu knihovnu. Je potřeba vzít v úvahu parametry plánovaného projektu a ty porovnat s možnostmi dané knihovny.

4 Objektová databáze

Objektové databáze ukládají data ve formě objektů. Takovéto objekty se zpravidla skládají z atributů (charakterizují objekt např. typy integer, string, apod.) a metod (definují chování objektu). Tennto přístup je výhodný z pohledu moderních objektově orientovaných programovacích jazyků, jako je C++, `C#`, Java a další. Objektové databáze dokáží ukládat komplexní objekty a vztahy mezi nimi, z pohledu vývojáře tedy odpadá nutnost převádět objektová (programová) data na něco jiného, jako je tomu v případě relačních databází. [14]

4.1 Eloquera

Eloquera je čistokrevná .NET objektová databáze vyvíjená stejnojmennou firmou. Dokáže nativně uložit jakýkoliv .NET objekt bez jakéhokoliv rozhraní nebo dědění ze speciálně navržené třídy. V rámci vývoje umí běžet v desktopovém módu ve Visual Studiu (jeden uživatel) a pochopitelně i v produkčním klient-server. Existuje ve dvou verzích - community edition (CE) a enterprise edition (EE). Community edice je zdarma pro osobní i komerční využití, cena enterprise verze se odvíjí podle zvolených parametrů. Rozdíl mezi nimi spočívá v lepší škálovatelnosti či load balancingu ve prospěch enterprise verze. [15]

4.1.1 Instalace

Doporučuji stáhnout Eloqueru přes NuGet, jelikož oficiální stránky (http://eloquera.com/download) vyžadují registraci, která je sice zdarma, ale někomu může vadit. Knihovna se do projektu nainstaluje v pohodě tří referencí, a to je prakticky vše, co je nezbytné pro začátek vývoje.

4.1.2 Technologie

Podobně jako některé dokumentové databáze, také Eloquera nevyžaduje žádné schéma. Už z principu jejího návrhu totiž žádné nepotřebuje - jedná se o objektovou databázi, která s objekty pracuje v jejich nativní formě. Narozdíl od relačních databází nepotřebuje při každém přesunu dat "rozsekávat"objekty a po kouscích je ukládat nazpátek do tabulky; zkrátka uloží objekt přesně tak, jak existuje v aplikaci.

Unikátní ID

Eloquera má svůj vlastní způsob generování unikátních identifikačních atributů. V prvé řadě vyžaduje proměnnou (nikoliv property) typu long, která je navíc označena anotací [ID]. Je potřeba zdůraznit, že takové ID není náhodná hodnota nebo inkrement, ale komplexní datový člen (field) se spoustou informací jako např. definice typu. Z toho důvodu jej vývojáři v žádném případě nedoporučují měnit.

Dotazovací API
Momentálně jsou k dispozici dvě - LINQ a SQL.

```
var user = (from User u in db where db.UID == idUser select u).Single();    //LINQ

Parameters param = db.CreateParameters();
param["Id"] = Id;
var user = (User)db.ExecuteQuery("SELECT␣User␣WHERE␣$ID␣=␣@Id", param); //SQL
```

Výpis 16: Eloquera - dotazování

V případě obou metod poporuje Eloquera také join syntaxi. Můžeme si všimnout poněkud zvláštního zápisu u podmínky ohledně ID. Je to z toho důvodu, že hodnota datového členu označeného [ID] neexistuje/není definována za hranicemi aplikace. `$ID` v SQL a db.UID v LINQu znamená dotaz prostřednictvím UID, což je unikátní ID objektu, podle kterého jej už knihovna najde. Obecně je SQL v Eloqueře robustnější než link a lze jeho prostřednictvím volat nejen datové členy a property, ale také potomky, pole a pole objektů.

Vývojové API
Jedna ze slabších stránek Eloquery, jelikož vyvíjet pro ni lze pouze na .NET platformě Microsoftu v jazyce `C#`.

Dynamické objekty
Tyto objekty odhalují členy jako např. vlastnosti nebo metody až při běhu programu, nikoliv při kompilaci. Díky tomu mohou dynamické objekty uchovávat struktury, které neodpovídají nějakému statickému typu či formátu. [17] Mezi jejich další výhody patří:

- Není potřeba řešit např. chybějící property, přizpůsobí se.
- Automaticky zjistí datový typ, také se postarají o indexování.
- Mohou obsahovat sebe navzájem.
- Dají se volně konvertovat na nativní objekty a zpět.

Mezi běžné využití patří např. registrace uživatele, evidence nebo aplikace, kde data mají proměnný počet atributů různých typů.

In-memory databáze
Eloquera podporuje databáze uložené v paměti, což je výhodné v případě, že je zapotřebí vysoká rychlost zpracování dat. Tuto technologii lze zapnout parametrem v connection stringu:

```
DB db = new DB("server=(local);options=inmemory,persist;");
```

Výpis 17: Eloquera - connection string

Parametr persist znamená, že se dababáze uloží na disk při volání db.Close(), což vytvoří soubor nesoucí její název. Takto uloženou databázi je možné opět otevřít příkazem db.OpenDatabase("název"). Server=(local) znamená, že databáze poběží v desktop módu.

Uložené procedury
Bloky kódu, které mohou být vyvolány ze strany databáze. Aplikace, které je využívají, mohou mít výhody:

- Snížená síťová zátež mezi klienty a servery - procedury operují s daty a vracejí pouze výsledky
- Zvýšená bezpečnost - se zahrnutím databázových práv do SP může administrátor zvýšit bezpečnost, jelikož všichni ostatní mohou využívat služeb procedury a nemusejí tak být sami vlastníci práv.
- Rychlejší vývoj - databázové operace se často opakují, tudíž znovuvyužití jedné procedury usnadňuje práci.

Procedury jsou v Eloqueře psané v `C#`. Deklarují se jako třídy dědící ze StoredProcedure.

4.1.3 Eloquera - shrnutí

Tato knihovna byla poprvé publikována 7.6.2010 ve verzi 2.7, současný build 6.3.3 pak 31.1.2014. Eloquera je objektová databáze, která se snaží odstranit největší nevýhody RDBMS systémů a zároveň programátorům nabídnout příjemné prostředí, ve kterém mohou s objekty zacházet bez toho, aniž by byli omezeni možnostmi relačních databází.

Eloquera nabízí řadu technik, jež s relačními databázemi zkrátka nejsou možné - ukládání nativních .NET objektů, dotazování na potomky/pole objektů/dle datového typu, velice dobře pracuje s komplexními objekty, podporuje dynamické objekty včetně kontejnerů atd.

Tvůrci o ní mluví jako o knihovně do enterprise sféry zaměřené na webové prostředí v rámci víceuživatelských systémů. V komunitní verzi je sice zdarma, nicméně enterprise verzi s plnou podporou nezbytných tenchnologií (profiling, disaster recovery apod.) je nutné zakoupit. Některým vývojářům také může vadit, že je Eloquera svázána s .NET platformou a žádný alternativní klient neexistuje, respektive nejspíše nikdy existovat nebude, jelikož se nejedná o open-source software a autoři tuto situaci nevidí jako problém.

Bohužel se mi z důvodu nedostatku času nepodařilo zpracovat druhou objektovou databázi, takže nemohu Eloqueru s nějakou další objektovou knihovnou přímo porovnat.

5 Scénáře persistence

V této závěrečné kapitole rozeberu možnosti a doporučené scénaře každé databázové technologie.

5.1 Relační databáze

Tyto databáze ukládají data do tabulek, které jsou rozděleny na sloupce, přičemž každý sloupec obsahuje jeden datový typ. Jeden záznam odpovídá jednomu řádku. Tabulky zpravidla obsahují klíče na jednom nebo více sloupcích, které pro každý řádek představují unikátní identifikátor. Nejjednodušším způsobem, jak spravovat data v relační databázi, je jazyk SQL.

Pokud tedy chce relační databáze uložit objekt, musí vytvořit jeho datovou reprezentaci. Ukládání objektů v takové databázi není jednoduché z důvodu nekompatibilních typů - je potřeba namapovat objektové schéma na relační databázi. Této technice se říká objektově-relační mapování a je nutné ji provádět při každém čtení/ukládání aplikačních dat do RDBMS. Z toho důvodu vznikly ORM řešení jako např. Entity Framework, jež vývojářům usnadňují překonat rozdíl mezi relačními a objektovými daty, nicméně tyto frameworky pouze zjednodušují psaní kódu, objektově-relační konflikt nevyřeší. Dále nejsou relační databáze vhodné pro ukládání dat s proměnnou strukturou (mají pevně dané schéma) a špatně škálují při obrovském množství dat.

Na druhou stranu mají výhodu v tom, že je to pochopená a zavedená technika, dobře se mapuje na data s konzistentní strukturou a větším množstvím vztahů mezi více entitami, umožňují joinování a podporují ACID transakce včetně jejich dobré škálovatelnosti. V enterprise segmentu se také hledí na to, že se jedná o velmi vyladěné technologie s podporou velkých korporací - spolehlivost.

Doporučené scénáře:

- Bankovnictví - střední dataset a rychlé transakční operace.
- Datové sklady - velký dataset, menší počet read/write operací s menším důrazem na rychlost.

5.2 Dokumentové databáze

NoSQL databáze představily nový způsob, jak ukládat data. V tomto modelu jsou každý záznam a jeho přidružená data chápáni jako dokument. Tyto dokumenty nemají pevně stanovenou strukturu (jsou bez schématu), díky čemuž mohou obsahovat poměrně komplexní data jako stromy, kolekce nebo slovníky. Na druhou stranu nepodporují relace; dokument může obsahovat klíč na jiný dokument, ale není zde žádná vynucená relační integrita.

Jelikož dokumenty nevyžadují žádné schéma, není nutné se zabývat konverzí objektových dat na relační a naopak. Také díky tomu nepředstavuje problém ukládat nestrukturovaná data a vyhnout se tak nákladným migracím. Dokumenty jsou navíc nezávislé

entity, což zvyšuje výkon (přidružená data jsou souvisle načítána z disku) a umožňuje jednoduše distribuovat data na vícero serverů. Tento typ databáze také dokáže velmi dobře vertikálně škálovat - s rostoucím počtem serverů téměř úměrne roste i výkon (velmi dobrá replikace a sharding).

Mezi nevýhody může patřit fakt, že dokumentové databáze zpravidla neumí join a často mají problémy zajistit ACID transakce. Povětšinou se navíc jedná o relativně mladé projekty, které svou spolehlivostí (myšleno vyzrálostí) nemohou odladěným RDBMS kokurovat.

Doporučené scénáře:

- Geo-indexing služby jako Foursquare či Craigslist - obrovský dataset s velkým množstvím jednodušších read/write operací.

5.3 Objektové databáze

Jak lze z názvu usoudit, objektové databáze ukládají data ve formě objektů, což jim dovoluje pracovat s aplikačními daty v jejich původní formě bez potřeby nějaké konverze. To znamená, že lze do databáze bez problému uložit celou kolekci objektů, dynamické objekty a velice složité objektové grafy obecně.

Podobně jako dokumentové databáze, ani ty objektové nepotřebují striktně definované schéma, což pro vývojáře znamená méně kódu a snadnější refaktoring. Dále mají zpravidla vysoký výkon (žádná konverze dat a většinou není potřeba join), lze porovnávat přímo podle identity objektu, dobře implementují dědičnost, zapouzdření, agregaci a polymorfismus.

Nicméně ve výsledku se jedná o mladou a nevyzrálou technologii. Na trhu zaujímá minimální podíl a často zkrátka stojí mimo okraj zájmu. V současnosti neexistuje žádný vendor, který by se mohl měřit s giganty jako Microsoft či Oracle, ale reálně ani s knihovnami jako MongoDB.

Doporučené scénáře:

- Případy komplexních dat nebo vztahů (many to many) - např. CAS/CAD/CAM aplikace.

6 Závěr

V této práci jsem dle typu mezi sebou porovnal 6 knihoven pro persistanci dat v rámci vývojové platformy Microsoft .NET a zhodnotil, jaký typ databáze je vhodný na ty které operace. Vůbec poprvé jsem se prakticky setkal s NoSQL a objektovými databázemi a do jisté míry se s nimi pomocí představených knihoven naučil pracovat, avšak nepodařilo se mi proniknout do dané problematiky tak hluboko, jak bych si představoval.

Ondřej Sobek

7 Reference

[1] *StackOverflow's ORM goes Open Source - Dapper.Net [online].* Dostupné z: http://www.infoq.com/news/2011/04/dapper-released

[2] *Stored Procedures (Database Engine) [online].* Dostupné z: http://msdn.microsoft.com/en-us/library/ms190782.aspx

[3] *Get Started with Entity Framework [online].* Dostupné z: http://msdn.microsoft.com/en-us/data/ee712907.aspx

[4] *Working with DbContext [online].* Dostupné z: http://msdn.microsoft.com/en-us/data/jj729737.aspx

[5] *Transactions (Database Engine) [online].* Dostupné z: http://technet.microsoft.com/en-us/library/ms190612(v=sql.105).aspx

[6] *Working with Transactions (EF6 Onwards) [online].* Dostupné z: http://msdn.microsoft.com/en-us/data/dn456843.aspx

[7] *NHibernate for .NET [online].* Dostupné z: http://msdn.microsoft.com/enus/library/bb399572.aspx

[8] *Use of implicit transactions is discouraged [online].* Dostupné z: http://www.hibernatingrhinos.com/products/nhprof/learn/alert/DoNotUseImplicitTransactions

[9] *NHibernate Forge [online].* Dostupné z: http://nhforge.org/

[10] *Second Level Cache for Entity Framework 6.1 [online].* Dostupné z: https://efcache.codeplex.com/

[11] *Databases: relational vs object vs graph vs document [online].* Dostupné z: http://www.cbsolution.net/techniques/ontarget/`databases_relational_vs_object_vs`

[12] *RavenDB Documentation [online].* Dostupné z: http://ravendb.net/docs

[13] *Redis Documentation [online].* Dostupné z: http://redis.io/documentation

[14] *Object Oriented Databases [online].* Dostupné z: http://www.comptechdoc.org/independent/database/basicdb/dataobject.html

[15] *Eloquera Licensing [online].* Dostupné z: http://eloquera.com/content/eloquera-licensing

[16] *Eloquera Documentation [online].* Dostupné z: http://wiki.eloquera.com/Eloquera-Database.ashx

[17] *Creating and Using Dynamic Objects [online].* Dostupné z: http://msdn.microsoft.com/en-us/library/vstudio/ee461504(v=vs.100).aspx